I2-IT224 Rapport
Programmation Multicoeur et GPU

Binôme
Étienne THIERY, Victor SAINT GUILHEM
Encadrant : *Raymond NAMYST*

Ce projet a pour but d'étudier le problème des tas de sable abéliens dont une présentation peut être trouvée sur cette page. Il est à noter que le modèle du tas de sable est ici simplifié selon des règles d'éboulement spécifiées.

1 Choix d'une structure de données

Nous avons commencé par développer en C, puis nous sommes tournés vers du C++ afin de factoriser le code. Ainsi, toute représentation d'un tas de sable et de son comportement doit implémenter l'interface `Sandpile`.

La classe abstraite `ArraySandpile` implémente les méthodes de cette interface concernant la structure de données à proprement parler (mais pas la méthode de simulation de l'écoulement).

Pour faciliter la parallélisation via OMP par la suite, et après quelques tentatives différentes peu fructueuses, nous nous sommes contentés de représenter la pile par un tableau à une dimension contenant les hauteurs des différentes cellules du tas de sable, ligne après ligne.

Un second tableau de même dimension est utilisé comme tableau de travail à chaque itération, avant d'être échangé avec le tableau représentant l'état actuel de la pile.

Plusieurs classes représentant différentes méthodes de calcul de l'écoulement du tas de sable partagent cette représentation, en héritant de cette classe abstraite. La hiérarchie finale du projet est représentée en figure 1 :

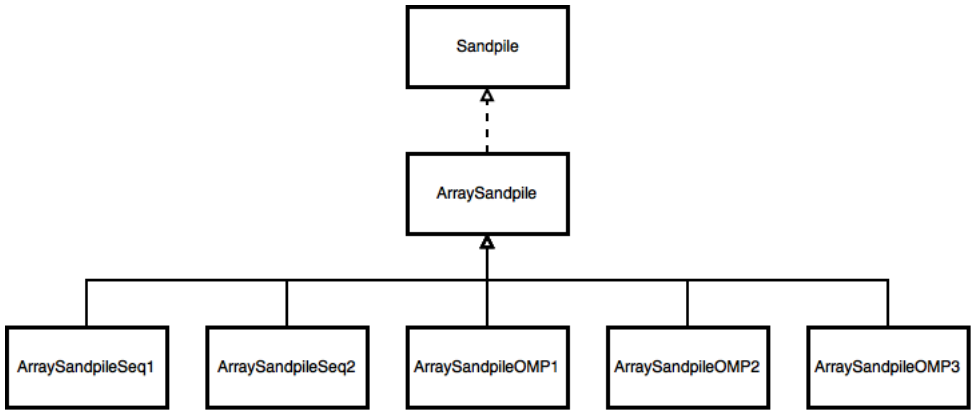

FIGURE 1 –

2 Solutions développées

2.1 Solutions séquentielles

Nous avons commencé par développer plusieurs solutions séquentielles, variantes des deux solutions conservées.

La première, la plus évidente, dérive directement de la règle d'écoulement du tas de sable. Elle consiste à considérer les cases une par une, et à les faire s'écrouler sur leur voisines. C'est la plus rapide des deux versions séquentielles mais, la valeur de chaque case étant modifiée jusqu'à 5 fois (sauf pour les cases du bord), cela rend difficile la parallélisation.

La seconde version séquentielle inverse le raisonnement : pour chaque case, elle lit les valeurs de toutes les cases voisines pour déterminer sa valeur au pas de temps suivant, et ne la modifier qu'une seule fois. Elle est plus lente, comme le montre la figure 2.1. Mais elle est beaucoup plus propice à la parallélisation, et c'est d'ailleurs sur elle que sont basées les versions OMP.

Une autre chose intéressante à remarquer est l'utilisation d'un peu de mémorisation sur certaines conditions (notamment savoir si une case est sur la bordure du tas de sable) pour accélérer le calcul.

version \ config	homogène 128	homogène 512	pic 128	pic 512
seq1	0,282	79	4,1	56
seq2	0,934	256	12,8	228
omp	0,15	10	1,9	19

FIGURE 2 – Comparaisons du temps de calcul des différentes versions (en secondes, avec le nombre de threads le plus adapté pour la version OMP)

2.2 Solutions parallèles

2.2.1 OMP

Nous avons développé plusieurs solutions parallélisées grâce à OMP.

La figure 2.2.1 représente la courbe de speedup pour la version ompOnly, développée en C hors de la hiérarchie de classe présentée précédemment. Les résultats obtenus sont assez satisfaisants puisque l'on a pu obtenir un speedup de 32 avec 64 coeurs sur la machine Jolicoeur !

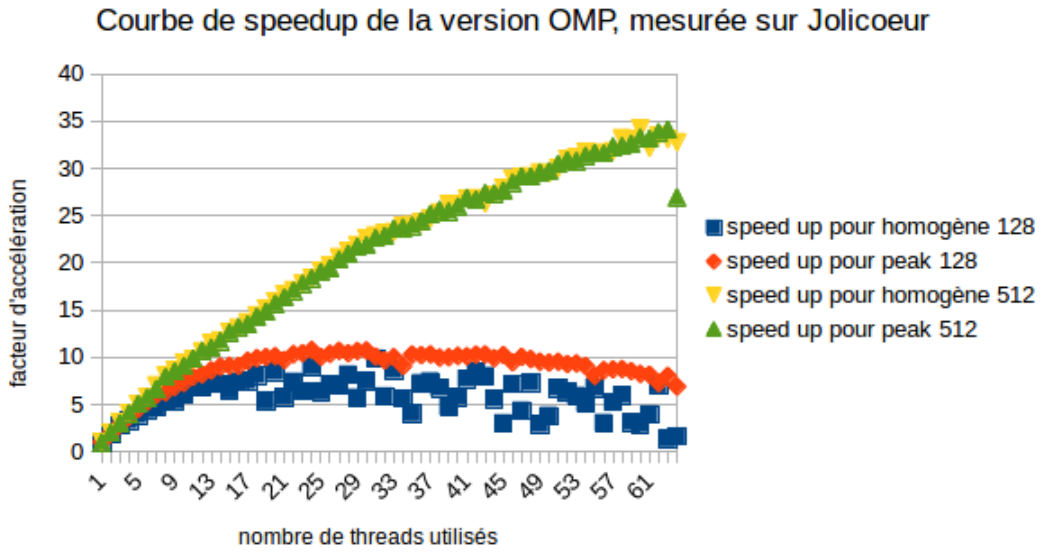

FIGURE 3 –

Les version omp1, omp2 et omp3 sont intégrées au même exécutable que les versions séquentielles mais ne fonctionnent pas encore tout à fait. On a un speed up de 1 à quelques coeurs mais cela se tasse beaucoup plus rapidement qu'ompOnly malheureusement.

La version omp1 est simplement la version parallélisée de la seconde version séquentielle, grâce à OMP, en découpant le tableau en N parts égales, où N est le nombre de threads. La version omp2 précalcule pour chaque case les cases susceptibles de lui « céder des grains de sable » pour accélérer le calcul. Enfin la version omp3 découpe le tas de sable différemment, en lignes complètes et consécutives afin de profiter des mécanismes de cache. Ce découpage facilite aussi le travail pour la version avec désynchronisation momentanée, puisqu'une boule de rayon R autour de L lignes correspond simplement à $2 * R + L$ lignes.

Nous avons compris et commencé à travailler sur la version où les threads se désynchronisent pendant quelques étapes, mais avec nos stages débutant nous n'avons malheuresement pas eu le temps de terminer.

2.2.2 GPU

Nous n'avons malheureusement pas eu le temps de travailler sur la parallélisation sur GPU.

2.3 Compilation et utilisation

Un simple make dans le répertoire du projet permet de tout compiler. L'éxecutable sable doit être appelé avec 4 arguments représentant respectivement la taille du tas de sable, la configuration initiale, le nombre d'étapes calculées à chaque appel de fonction et la quantité d'information à afficher (voir la doc de l'exécutable en exécutant seulement `./sable`) :

```
./sable -s 128 -c 0 -b 10 -d 1
```